

**JU Srednja elektrotehnička škola Mostar**

## **MATURSKI RAD**

### **Tema: Dizajn i razvoj video igara**

Članovi komisije:

1. \_\_\_\_\_, predsjednik
2. \_\_\_\_\_, član
3. \_\_\_\_\_, član

Ocjena: \_\_\_\_\_

Predmetni profesor:

*Sena Letuka, prof.*

Učenik:

*Admer Šuko, IV<sub>1</sub>*

**Mostar, april 2021.**

# Sadržaj:

1. Uvod.....	1
2. Game engine.....	2
2.1. Struktura engine-a.....	2
2.1.1. Sistemski interfejs.....	2
2.1.2. Zajednička biblioteka qcommon .....	3
2.1.3. Zajednička biblioteka bLib .....	4
2.1.4. Serverski modul.....	5
2.1.5. Klijentski modul.....	6
3. Igra i razvoj sistema igre .....	7
3.1. Struktura igre.....	7
3.1.1. Folderska struktura igre.....	7
3.1.2. Kodna struktura igre .....	9
3.2. Implementacija entitetskog sistema .....	11
3.2.1. Proceduralni entitetski sistem .....	11
3.2.2. Objektno orijentisani entitetski sistem.....	15
3.2.3. Pregled klasne hijerarhije .....	19
3.3. Implementacija sistema za oružje .....	20
3.3.1. Model razdvojenih klasa za klijent i server.....	20
3.3.2. Interpretiranje interakcija s oružjem.....	21
3.3.3. Vršenje napada .....	24
3.4. Igrač.....	25
3.4.1. Implementacija igrača na klijentu.....	25
3.4.2. Implementacija igrača na serveru.....	28
3.5. AI sistem .....	29
3.5.1. Snalaženje u prostoru.....	29
3.5.2. Pamćenje.....	30
3.5.3. Detekcija neprijatelja .....	32
3.5.4. Evaluacija situacije .....	33
3.5.5. Klasifikacija prijatelja i neprijatelja .....	34

3.6. Pravila igre i brojnost igrača .....	36
3.6.1. Singleplayer .....	36
3.6.2. Multiplayer .....	36
3.6.3. Kooperacija.....	36
4. Sadržaj i dizajn igre.....	37
4.1. Osnovni podaci .....	37
4.2. Izrada 3D modela.....	38
4.3. Izrada levela.....	40
4.4. Izrada muzike.....	41
4.5. Izrada tekstura .....	41
Zaključak .....	42
Prilog .....	43
Terminologija .....	43
Literatura .....	44

# 1. Uvod

Dizajn i razvoj igara dio je opširnog, kompleksnog procesa izrade video igara. U kontekstu video igara, dizajn predstavlja osmišljenje i implementaciju mehanika, levela i likova igre, odnosno svih njenih logičkih i interaktivnih aspekata; dok razvoj predstavlja samu izradu sadržaja igre: 3D modela predmeta, muzike, zvučnih efektata; kao i programiranje igre, koje može uključivati programiranje umjetne inteligencije, mrežno programiranje, programiranje entitetskog sistema itd.

Tokom razvoja igre, zavisno od vrste resursa koji se stvara, koriste se različite vrste alata: IDE (razvojno okruženje za programiranje), level editor, model editor, editor slika, audio editor, tekst editor itd. Tehnologija koja učitava resurse i izvršava kod od igre zove se, doslovno prevedeno, motor igre (eng. *game engine*), odnosno **engine**.

U Radu (ovom radu) obrađuju se - ne nužno u navedenom redu - temeljni aspekti razvoja igre: game engine koji igra koristi, dizajn mehanika, dizajn levela, struktura igre, implementacija entitetskog sistema za objekte u igri, implementacija umjetne inteligencije; a isti se upotpunjuju sa suplementnim aspektima, uključujući i korištene alate i biblioteke.

Video igre su vrlo vizuelna i interaktivna vrsta softvera, i prema tome, radi lakšeg shvatanja ideja, u radu se nalazi mnogo ilustracija i slika, kao i referentni primjerni kod. Među posljednjim stranicama ovog rada nalazi se rječnik sa svim ključnim definicijama.

**Ključne riječi i fraze:** game engine, video igra, razvoj, dizajn, level, 3D model, entitetski sistem, umjetna inteligencija, open-source

## 2. Game engine

Game engine je polazna tehnologija za igru, koja znatno ubrzava i olakšava razvojni proces iste. Na tržištu postoji širok izbor engine-a<sup>1</sup>, kao što su *Unreal Engine 4*, *Unity*, *CryEngine V*, *Godot Engine*, *Source Engine*, *Lumia Engine* itd.

Zbog potreba Rada, odabran je *BUREKTech*<sup>2</sup> engine, koji je nastao kao preinačena verzija *ioquake3* engine-a. *ioquake3* nastao je kao preinačena verzija od *idTech 3*. *idTech 3* razvila je firma *id Software* 1999. godine sa kojim je, iste godine, napravila vrlo popularnu igru *Quake 3 Arena*. Kasnije, studija za razvoj igara su licencirala taj engine i sa njim su razvila igre kao što su *Medal of Honor: Allied Assault* (2002, 2015 Inc. – *Electronic Arts*), *Call of Duty* (2003, *Infinity Ward* – *Activision*) i *Return to Castle Wolfenstein* (2001, *Gray Matter Interactive* – *Activision*).

*idTech 3*, prema tome i *ioquake3*, napisan je u programskom jeziku C. Zbog mogućnosti objektno-orijentisanog programiranja, *BUREKTech* je preveden u C++. Većina koda je identična *ioquake3*, osim što se za pojedine komponente koriste klase, interfejsi, `std::vector` i slično. Detaljniji pregled engine-a, njegova struktura i osnovne funkcionalnosti prikazani su u sljedećim dijelovima.

### 2.1. Struktura engine-a

Engine se sastoji od sljedećih modula:

- sistemski interfejs (sys)
- zajednička biblioteka 1 (qcommon)
- zajednička biblioteka 2 (bLib)
- serverski modul (server)
- klijentski modul (client)

#### 2.1.1. Sistemski interfejs

**Sistemski interfejs** je komponenta najnižeg nivoa. Sadrži razne sistemske funkcije kao što je ulazna tačka programa, funkcija za učitavanje DLL fajla, funkcija za restart aplikacije itd. Engine koristi biblioteku *SDL2* da bi se znatno olakšala podrška za više platformi, tako da je sistemski interfejs daleko više generičan i čist.

---

<sup>1</sup> Popis game engine-a, Wikipedia - [https://en.wikipedia.org/wiki/List\\_of\\_game\\_engines](https://en.wikipedia.org/wiki/List_of_game_engines)

<sup>2</sup> *BUREKTech* game engine, Admer Šuko - <https://github.com/Admer456/ioq3-burek/tree/game/cirkuz33>

## 2.1.2. Zajednička biblioteka *qcommon*

*qcommon* sadrži najosnovnije definicije i generične funkcije koje se često koriste kako u engine-u, tako i u igri. Unutar header fajla *q\_shared.hpp*, naprimjer, nalazi se sljedeća struktura:

```
1329 // usercmd_t is sent to the server each client frame
1330 typedef struct usercmd_s {
1331     int         serverTime;
1332     int         angles[3];
1333     int         buttons;           // general-purpose buttons; 32 bits
1334     short       interactionButtons; // special-purpose buttons; 16 bits
1335     byte        weapon;           // weapon
1336     signed char forwardmove, rightmove, upmove;
1337 } usercmd_t;
1338
```

Slika 1. Struktura *usercmd\_t*

*usercmd\_t* je struktura koja je specifična za igrače, a skraćenica je od „user commands“, značeći „korisničke naredbe“. Ukoliko igrač upre tipku na tastaturi za kretanje naprijed, varijabla *forwardmove* će poprimiti maksimalnu vrijednost 127. To se dalje može iskoristiti u kodu za kretanje, no i za ostale potrebe. Korisničke komande periodično šalju klijenti prema serveru, a server iste prosljeđuje igri.

*qcommon* također sadrži dio mrežnog koda, koji koristi serverski modul:

```
954 netField_t  entityStateFields[] =
955 {
956     // pos
957     netField_t{ NETF(pos.trType), 8 },
958     netField_t{ NETF(pos.trTime), 32 },
959     netField_t{ NETF(pos.trDuration), 32 },
960     netField_t{ NETF(pos.trBase[0]), 0 },
961     netField_t{ NETF(pos.trBase[1]), 0 },
962     netField_t{ NETF(pos.trBase[2]), 0 },
963     netField_t{ NETF(pos.trDelta[0]), 0 },
964     netField_t{ NETF(pos.trDelta[1]), 0 },
965     netField_t{ NETF(pos.trDelta[2]), 0 },
966
```

Slika 2. Mrežno enkodiranje varijabli za objekte u igri

Na slici 2 prikazano je prvih devet zajedničkih entitetskih polja koja se enkodiraju i šalju klijentima. Svaki objekat u igri ima osnovna svojstva kao što su pozicija, orijentacija, model, naziv, indeks itd. Svako polje se enkodira različitim brojem bitova, npr. *pos.trType* se enkodira s 8 bitova, dok se *pos.trTime* enkodira sa 32 bita.

### 2.1.3. Zajednička biblioteka bLib

*bLib* je biblioteka specifična za *BUREKTech*. *qcommon* bio je prisutan već od izvornog *idTech 3* engine-a, za razliku od *bLib*-a. *bLib* je napisan u modernom C++-u, što podrazumijeva klase, namespace-ove i template, dok je *qcommon* napisan u C stilu, što podrazumijeva strukture i globalne funkcije.

Klasa koja se najviše koristi iz *bLib*-a je *Vector*. *Vector* je matematička klasa koja predstavlja 3D vektor, te razne metode i operatore za manipulaciju i računanje istih. Prototip klase prikazan je na slici 3.

```
11 // We don't care about converting from double to float
12 #pragma warning( disable : 4244 )
13
14 class Vector
15 {
16 public:
17     Vector() { x = y = z = 0; }
18
19     // Basic way of making a vector
20     Vector( float X, float Y, float Z ) { x = X; y = Y; z = Z; }
21
22     // Vector from another vector
23     Vector( const Vector& v ) { x = v.x; y = v.y; z = v.z; }
24
25     // vec3_t support
26     Vector( float* vec ) { x = vec[0]; y = vec[1]; z = vec[2]; }
27     Vector( const float* vec ) { x = vec[0]; y = vec[1]; z = vec[2]; }
28
29 public: // Utilities
30
31     // Length of the vector
32     inline float Length() const
33     {
34         return sqrt( static_cast<float>( x*x + y*y + z*z ) );
35     }
```

Slika 3. Klasa Vector

*Vector* sadrži 271 liniju koda, a njegova implementacija sadrži 140, tako da se neće prikazati sve funkcije ove klase. Na dnu slike prisutna je *Length* metoda, koja računa dužinu vektora prema sljedećoj matematičkoj formuli:

$$|v| = \sqrt{x^2 + y^2 + z^2}$$

## 2.1.4. Serverski modul

Serverski modul, skraćeno **server**, je komponenta engine-a koja učitava DLL od igre i izvršava ga, periodično emituje enkodirane podatke klijentu, i asinhrono manifestuje vezu sa klijentima. Server koristi vlastito rješenje za emitovanje datagrama preko mreže, oslanjajući se na *sendto* funkciju, koja direktno šalje datagram na određeni socket klijenta.

Prilikom pokretanja servera, odnosno kad igrač pokrene level iz glavnog izbornika, učitava se igrin DLL u funkciji *SV\_GameInitVM*:

```
893  /*
894  =====
895  SV_InitGameVM
896  -----
897  Called for both a full init and a restart
898  =====
899  */
900  static void SV_InitGameVM( qboolean restart ) {
901      int      i;
902
903      // start the entity parsing at the beginning
904      sv.entityParsePoint = CM_EntityString();
905
906      // clear all gentity pointers that might still be set from
907      // a previous level
908      // https://zerowing.idsoftware.com/bugzilla/show\_bug.cgi?id=522
909      // now done before GAME_INIT call
910      for ( i = 0 ; i < sv_maxclients->integer ; i++ ) {
911          svs.clients[i].gentity = nullptr;
912      }
913
914      // use the current msec count for a random seed
915      // init for this gamestate
916      game->Init( sv.time, Com_Milliseconds(), restart );
917  }
```

Slika 4. Inicijalizacija igre

Objekat *game* služi kao interfejs između engine-a i igre.



### 2.1.5.      **Klijentski modul**

Klijentski modul, skraćeno **klijent**, je komponenta engine-a koja prima i procesuirá datagrame od servera, te korisniku **prezentira podatke**, i serveru šalje podatke o igračevim postupcima. U drugim riječima, klijent prikazuje vidljive entitete igraču, kao i level u kojem se nalazi, a serveru šalje komande koje igrač želi da izvrši (npr. kretanje).

Za vizuelnu prezentaciju korisniku, klijent učitava takozvani *renderer DLL*, koji implementira određene funkcije za grafički prikaz sadržaja, odnosno renderovanje. Renderer mora implementirati broj funkcija: registracija fontova, registracija 3D modela, registracija 2D slika, renderovanje modela, računanje svjetla za datu poziciju itd.

Osim vizuelne prezentacije korisniku, klijent korisniku prezentira i zvuk. Zvučni sistem, poput renderer-a, je zaseban DLL fajl koji implementira sljedeće funkcije: registracija zvučnog fajla, reprodukcija muzike, zaustavljanje muzike itd.

## 3. Igra i razvoj sistema igre

Igra, odnosno video igra, je interaktivna vrsta digitalnog medija koji služi za zabavu, simulaciju, obrazovanje i/ili vježbanje. Zabavne video igre su često slične animiranim filmovima, međutim, razlika je što su video igre interaktivne, a filmovi nisu. Interakcija podrazumijeva korisnikovu sposobnost da utječe na događaje unutar igre, odnosno da unutar igre koristi predmete i vrši međudjelovanje sa likovima ili okolinom.

### 3.1. Struktura igre

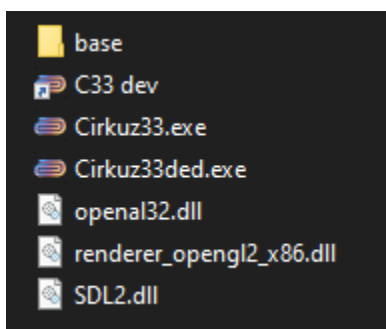
U kontekstu programskog koda, igra može biti struktuirana na različite načine. Struktura može zavisiti od toga da li je igra singleplayer (jedan igrač), multiplayer (više igrača) ili local multiplayer (više igrača na istoj mašini). Igra u sklopu Rada je struktuirana po multiplayer modelu, što znači da podržava igranje sa više igrača preko Interneta.

Struktura igre se može odnositi na strukturu fajlova u izvornom kodu, ili na strukturu foldera i fajlova u instalacijskom folderu igre.

#### 3.1.1. Folderska struktura igre

Cijeli sadržaj igre, poslije instalacije, nalazi se u jednom folderu. Taj folder naziva se korjenski folder igre. U korjenskom folderu nalaze se dva izvršna fajla: [igra].exe i [igra]ded.exe, gdje [igra] može biti bilo koji naziv igre. Prvi fajl je pokretač igre, namijenjen za igrače, koji će krajnji korisnik otvarati pomoću desktop prečaca. Drugi fajl namijenjen je za vlasnike servera, skoro je isti kao pokretač igre, osim što isključuje klijentsku komponentu, i zamjenjuje vizuelno predstavljanje igrinog sadržaja sa tekstualnim interfejsom.

Pored pokretača igre, nalazi se renderer DLL, framework DLL i DLL zvučnog sistema, prikazano na slici ispod:

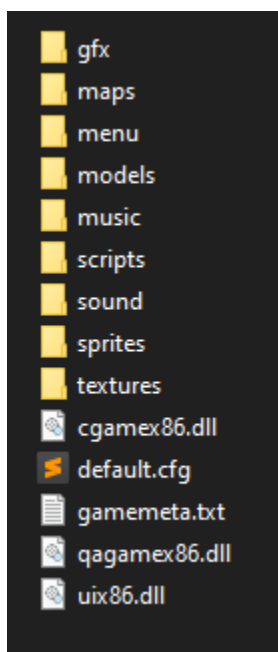


Slika 5. Korjenski folder igre

openal32.dll je biblioteka zvučnog sistema *OpenAL-soft*<sup>3</sup>. renderer\_opengl2 je grafički sistem *BUREKTech* engine-a. SDL2.dll je biblioteka multimedijskog frameworka *SDL2*<sup>4</sup>.

Važno je napomenuti da su navedeni EXE fajlovi izvršni fajlovi game engine-a. Game engine, tokom pokretanja, učitava DLL fajlove od igre, koji se nalaze u jednoj od direktorija unutar korjenskog foldera. Takvi direktoriji se nazivaju „folderi igara“, gdje jedan direktorij predstavlja jednu zasebnu igru.

Da bi se game engine uspješno pokrenuo, potreban je bazni folder, sa nazivom „base“, unutar kojeg se nalaze ključni fajlovi, konfiguracije i skripte, kao na slici 6.



**Slika 6. Bazni folder igre**

Opis fajlova i foldera je sljedeći:

- gfx sadrži najosnovnije slike za 2D grafike unutar igre, uključujući fontove
- maps sadrži fajlove levela
- menu sadrži slike za izbornike unutar igre
- models sadrži 3D modele igre
- music sadrži muziku igre
- scripts sadrži skripte igre i skripte za specijalne grafičke efekte
- sound sadrži sve zvučne efekte igre

---

<sup>3</sup> OpenAL Soft, open-source implementacija OpenAL standarda - <https://openal-soft.org/>

<sup>4</sup> Simple DirectMedia Layer - <https://www.libsdl.org/index.php>

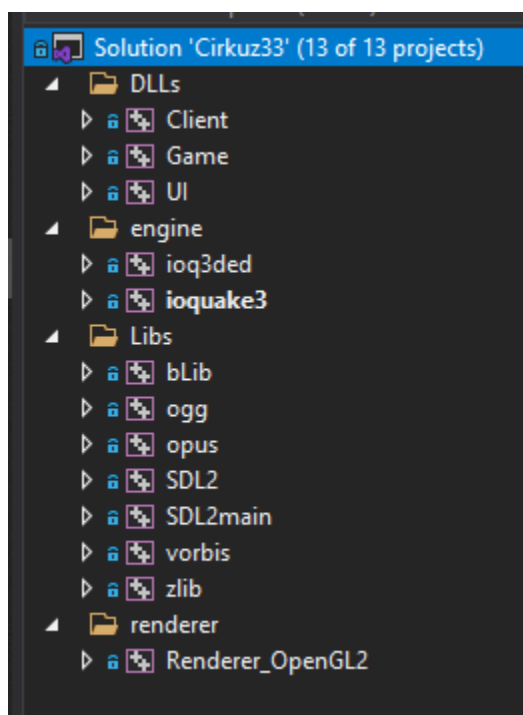
- sprites sadrži posebnu vrstu slika koje se prikazuju kao 2D grafike unutar igre
- textures sadrži sve slike koje se primjenjuju na 3D modele i levele igre
- cgame[x86/x64].dll je klijentski DLL od igre
- qagame[x86/x64].dll je serverski DLL od igre
- ui[x86/x64].dll je DLL korisničkog interfejsa igre
- default.cfg je uobičajena konfiguracija igre, koja mapira kontrole za tastaturu i miš po prvom pokretanju igre, kao i grafičke postavke i druge
- gamemeta.txt je tekstualni fajl koji sadrži metapodatke o igri, kao što su naziv igre, godina izdavanja, razvojna firma igre, izdavač itd.

### 3.1.2. Kodna struktura igre

Iz prethodnog dijela evidentno je da postoje tri različita DLL fajla koja pripadaju igri. Prema tome, polazna kodna struktura igre se sastoji iz tri glavne module:

- klijentska igra
- serverska igra
- korisnički interfejs igre

Na slici ispod, prikazani su Visual Studio projekti od cjelokupnog koda, koji uključuje igru, engine i biblioteke:



Slika 7. Visual Studio projekat igre

### 3.1.2.1. Klijentska igra

Klijentska igra prihvata vidljive entitete od klijentskog modula engine-a, te ih dodatno procesira na način specifičan za igru, te poziva engine da iste proslijedi u renderer za prezentaciju korisniku. Klijentska igra može da proslijedi dodatne entitete na isti način, da bi se ostvarili dodatni vizuelni efekti, ili čak implementirao sistem za vegetaciju.

Klijentska igra se sastoji od sljedećih komponenti:

- EventHandler sloja
- ClientView sloja
- mrežnog predikcijskog sistema
  - predikcije za oružje
  - predikcije za stanje igrača
  - predikcije za generične entitete
- interfejsa sa engine-om

Važnije navedene komponente će biti razrađene na narednim stranicama.

### 3.1.2.2. Serverska igra

Serverska igra implementira svu logiku vezanu za mehanike igre, kao i sve entitetske klase. Pored toga, implementira pravila igre, kao i interfejs igre sa kojim engine vrši komunikaciju.

Serverska igra se sastoji od sljedećih komponenti:

- entitetskog sistema zajedno sa entitetskim klasama – oblast 3.2
  - klase za igrače (BasePlayer) – oblast 3.4.2
  - bazne entitetske klase (IEntity, BaseEntity)
  - entitetskih klasa za oružje
  - ostalih entitetskih klasa
- sistema za umjetnu inteligenciju (AI) – oblast 3.5
- sistema za emitovanje eventova, koje prihvata EventHandler sloj u klijentu
- interfejsa sa engine-om

Važnije navedene komponente, uključujući implementaciju entitetskog sistema, će biti razrađene na narednim stranicama.

### 3.1.2.3. Korisnički interfejs igre

Korisnički interfejs igre implementira sve aspekte korisničkog interfejsa u igri, kao što su svi izbornici, stilovi, boje, komande u izbornicima, te pauzni ekran igre. Sastoji se od sljedećih komponenti:

- izbornika
- pomoćnih funkcija
- interfejsa sa engine-om

## 3.2. Implementacija entitetskog sistema

Entiteti u igri su objekti koji služe određenu funkciju. Cijeli level je sam po sebi entitet, koji služi funkciju prikazivanja igrivog svijeta igraču. Igrač je, u perspektivi igre, također entitet. Engine poznaje jedinstven entitetski interfejs (klasa  *IEntity*), sa kojim rukuje u serverskom modulu, dok igra poznaje implementaciju entiteta (klasa  *BaseEntity*).

### 3.2.1. Proceduralni entitetski sistem

U *ioquake3*, entitetski sistem nije bio objektno orijentisan, već proceduralan. Primjer funkcije za stvaranje jedne entitetske „klase“ je prikazan na slici ispod:

```
/*QUAKED trigger_multiple (.5 .5 .5) ? RED_ONLY BLUE_ONLY
"wait" : Seconds between triggerings, 0.5 default, -1 = one time only.
"random"    wait variance, default is 0
Variable sized repeatable trigger. Must be targeted at one or more entities.
so, the basic time between firing is a random time between
(wait - random) and (wait + random)
*/
void SP_trigger_multiple( gentity_t *ent ) {
    G_SpawnFloat( "wait", "0.5", &ent->wait );
    G_SpawnFloat( "random", "0", &ent->random );

    if ( ent->random ≥ ent->wait && ent->wait ≥ 0 ) {
        ent->random = ent->wait - FRAMETIME;
        G_Printf( "trigger_multiple has random ≥ wait\n" );
    }

    ent->touch = Touch_Multi;
    ent->use = Use_Multi;

    InitTrigger( ent );
    trap_LinkEntity (ent);
}
```

Slika 8. „Spawn“ procedura entiteta „trigger\_multiple“

Iz navedenog primjera, instanca „ent“ će poprimiti vrijednosti za nekoliko varijabli, među kojima su „touch“ i „use“. *Touch\_Multi* i *Use\_Multi* su globalne funkcije koje se nalaze iznad funkcije *SP\_trigger\_multiple*.

Pregled strukture *gentity\_t*<sup>5</sup> prikazan je na slikama ispod:

```

struct gentity_s {
    entityState_t  s;           // communicated by server to clients
    entityShared_t  r;           // shared by both the server system and game

    // DO NOT MODIFY ANYTHING ABOVE THIS, THE SERVER
    // EXPECTS THE FIELDS IN THAT ORDER!
    //=====

    struct gclient_s  *client;    // NULL if not a client

    qboolean  inuse;

    char      *classname;        // set in QuakeEd
    int       spawnflags;        // set in QuakeEd

    qboolean  neverFree;         // if true, FreeEntity will only unlink
    // bodyque uses this

    int       flags;             // FL_* variables

    char      *model;
    char      *model2;
    int       freetime;          // level.time when the object was freed

    int       eventTime;         // events will be cleared EVENT_VALID_MSEC after set
    qboolean  freeAfterEvent;
    qboolean  unlinkAfterEvent;

    qboolean  physicsObject;     // if true, it can be pushed by movers and fall off edges
    // all game items are physicsObjects,

    float     physicsBounce;     // 1.0 = continuous bounce, 0.0 = no bounce
    int       clipmask;          // brushes with this content value will be collided against
    // when moving.  items and corpses do not collide against
    // players, for instance

```

Slika 9. Početak strukture *gentity\_t*

Važno je spomenuti dvije strukture pri vrhu, *entityState* i *entityShared*. *entityState* sadrži osnovne varijable stanja entiteta, kao što su pozicija, rotacija, indeks entiteta, vrsta kretanja, indeks modela, dodatni podaci itd. Ovaj dio server prenosi klijentima. *entityShared* sadrži varijable koje zanima serverski modul engine-a, kao što su trenutno područje u kojem se entitet nalazi, sadržaji zapremine koju entitet obuhvata, dimenzije granične kutije itd.

<sup>5</sup> Skraćenica od „game entity type“

```
char      *target;
char      *targetname;
char      *team;
char      *targetShaderName;
char      *targetShaderNewName;
gentity_t *target_ent;

float     speed;
vec3_t    movedir;

int       nextthink;
void      (*think)(gentity_t *self);
void      (*reached)(gentity_t *self);    // movers call this when hitting endpoint
void      (*blocked)(gentity_t *self, gentity_t *other);
void      (*touch)(gentity_t *self, gentity_t *other, trace_t *trace);
void      (*use)(gentity_t *self, gentity_t *other, gentity_t *activator);
void      (*pain)(gentity_t *self, gentity_t *attacker, int damage);
void      (*die)(gentity_t *self, gentity_t *inflictor, gentity_t *attacker, int damage, int mod);

int       pain_debounce_time;
int       fly_sound_debounce_time;    // wind tunnel
int       last_move_time;

int       health;

qboolean  takedamage;
```

Slika 10. Relaciona polja entiteta; pokazivači funkcija

Svaka instanca entiteta može da ima svoj naziv (targetname), i metu (target). Neke posebne vrste entiteta, kao što su tipke, moraju imati svoju metu da bi mogli međudjelovati s istom. Najjednostavniji primjer mete su vrata. Ukoliko igrač pritisne fizičku tipku X u levelu, vrata Y će se otvoriti.

### 3.2.1.1. Međudjelovanje entiteta

Međudjelovanje entiteta se u kodu naziva korištenje (use, using), dok se u level dizajnu naziva **okidanje** (trigger, triggering). Korištenje se vrši na način da entitet nađe svoju metu pomoću naziva, odnosno dobije pokazivač na taj entitet, pa pozove njegovu use funkciju.

Međudjelovanje entiteta nije ograničeno samo na korištenje. Entiteti mogu međudjelovati na sljedeće načine:

- korištenje
- razmišljanje
- blokiranje
- dodirivanje

Razmišljanje je vrsta djelovanja entiteta gdje entitet periodično poziva svoju think funkciju. Primjer ovakvog entiteta je alarmno svijetlo, koje se konstantno okreće, odnosno periodično mijenja svoju rotaciju.

Blokiranje se primjenjuje na entitete koji imaju asocirani model i zapreminu. Ukoliko se voz kreće određenom putanjom, a na toj putanji se sudari sa drugim vozom,



kažemo da je jedan entitet blokirao drugog, te se u drugog entiteta poziva blocked funkcija.

Dodirivanje se primjenjuje na entitete koji imaju zapreminu, ali ne nužno model. Razni predmeti (hrana, alati, oružje, medicinska sredstva) imaju nevidljivu kutiju oko sebe koja se koristi za detekciju dodira s ostalim entitetima. Tako da, ukoliko igrač dodirne takvu kutiju, on poziva touch funkciju tog entiteta.

Prema tome, definisana je sljedeći skup imena:

- okidač – entitet koji okida svoju metu
- meta – entitet na kojeg pokazuje drugi entitet
- aktivator – entitet koji je prvi pokrenuo niz okidanja
- pozivnik – entitet koji je pozvao use funkciju svoje mete, ne nužno aktivator

U praktičnom primjeru, postoje igrač, tipka i vrata. Kad igrač iskoristi tipku, tipka mu postaje meta, a igrač je njen okidač, aktivator i pozivnik. Tipka potom okida vrata, te postaje okidač i pozivnik tih vrata. Međutim, aktivator vrata je i dalje igrač, pošto je započeo lanac okidanja. Lanci okidanja se alternativno nazivaju sekvence okidanja (eng. *trigger sequences*). Sekvence okidanja mogu biti velike, a broj entiteta koji mogu učestvovati u takvoj sekvenci je teoretski neograničen.

Primjer okidanja se nalazi na slici ispod:

```

235 void G_UseTargets( gentity_t *ent, gentity_t *activator ) {
236     gentity_t      *t;
237
238     if ( !ent ) {
239         return;
240     }
241
242     if (ent->targetShaderName && ent->targetShaderNewName) {
243         float f = level.time * 0.001;
244         AddRemap(ent->targetShaderName, ent->targetShaderNewName, f);
245         trap_SetConfigstring(CS_SHADERSTATE, BuildShaderStateConfig());
246     }
247
248     if ( !ent->target ) {
249         return;
250     }
251
252     t = NULL;
253     while ( (t = G_Find(t, FOFS(targetname), ent->target)) != NULL ) {
254         if ( t == ent ) {
255             G_Printf ("WARNING: Entity used itself.\n");
256         } else {
257             if ( t->use ) {
258                 t->use (t, ent, activator);
259             }
260
261             if ( !ent->inuse ) {
262                 G_Printf("entity was removed while using targets\n");
263                 return;
264             }
265         }
266     }
}

```

Slika 11. Pomoćna funkcija za okidanje entiteta

Parametar *ent* je pozivnik, dok je parametar *activator* aktivator. Funkcija *G\_Find* će tražiti sljedeći entitet u globalnom nizu entiteta dok se ne nađe nijedan drugi entitet s istim imenom.

### 3.2.2. Objektno orijentisani entitetski sistem

U *BUREKTech-u*, stari proceduralni sistem zamijenjen je OO entitetskim sistemom. DLL serverske igre, usljed toga, je skoro u potpunosti izmijenjen. Interfejs sa engine-om je zasnovan na C++ interfejsima i implementaciji istih, dok entiteti nisu više strukture, umjesto toga su različite klase.

#### 3.2.2.1. Entitetski interfejs

Implementacija objektno-orijentisanog entitetskog sistema započinje sa jedinstvenim entitetskim interfejsom, čiji je prototip prikazan na slici ispod:

```

namespace Entities
{
    class IEntity;

    typedef void (IEntity::* thinkPointer)    )( void );
    typedef void (IEntity::* usePointer)      )( IEntity* activator, IEntity* caller, float value );
    typedef void (IEntity::* touchPointer)    )( IEntity* other, trace_t* trace );
    typedef void (IEntity::* blockedPointer)  )( IEntity* other );
    typedef void (IEntity::* takeDamagePointer)( IEntity* attacker, IEntity* inflictor, int damageFlags, float damage );
    typedef void (IEntity::* diePointer)      )( IEntity* killer );

    class IEntity
    {
    public:
        virtual void        Spawn() = 0; // Gets called *while* entities are spawning
        virtual void        PostSpawn() = 0; // Gets called after all entities have spawned

        virtual void        Precache() = 0;
        virtual void        ParseKeyvalues() = 0;

        virtual void        Think() = 0;
        virtual void        Use( IEntity* activator, IEntity* caller, float value ) = 0;
        virtual void        Touch( IEntity* other, trace_t* trace ) = 0;
        virtual void        Blocked( IEntity* other ) = 0;
        virtual void        TakeDamage( IEntity* attacker, IEntity* inflictor, int damageFlags, float damage ) = 0;
        virtual void        Die( IEntity* killer ) = 0;

        virtual void        Remove() = 0; // Mark for removal

        virtual void        OnClientBegin( int clientNum ) = 0; // Called when the client joins and spawns
        virtual void        OnClientDisconnect( int clientNum ) = 0; // Called when the client disconnects

        virtual void        OnPlayerDie( int clientNum ) = 0; // Called when a player dies

        // All entities have an entity index
        virtual unsigned int GetEntityIndex() const = 0;
        virtual void        SetEntityIndex( const size_t& index ) = 0;

        constexpr static size_t EntityIndexNotSet = 1 << 31U;
    };
}

```

Slika 12. Prototip interfejsa IEntity

Bilo koja entitetska klasa bi, prema tome, morala implementirati sve navedene funkcije. Zbog toga je napisana bazna entitetska klasa, koja engine-u nije poznata, a sve njene funkcije implementirane su u DLL-u serverske igre.

Objašnjenja najvažnijih funkcija su sljedeća:

**Spawn** funkcija se poziva nakon alokacije, najčešće kad se level učitava i kad se dinamično stvara nova instanca entitetske klase usljed igranja igre.

**PostSpawn** funkcija specifična je za učitavanje levela. Nakon što se učitao level i „spawn-ovali“ svi entiteti, poziva se PostSpawn da bi entitet mogao dobiti pokazivače na svoje mete, između ostalog. Nepoželjno je tražiti pokazivače na entite unutar Spawn funkcije, zato što ima šansa da traženi entitet u datom momentu ne postoji u memoriji.

**Precache** funkcija govori engine-u koje fajlove je potrebno učitati. Naprimjer, klasa Automobil bi učitala 3D model od automobila, 3D model točkova, i zvukove motora.

**ParseKeyValues** učitava podatke iz levela, te dodjeljuje svojstva entiteta iz levela u varijable entiteta. U fajlu levela postoji dio zapisa koji se odnosi na entitete, u kojem su zabilježeni osnovni podaci (pozicija, naziv klase, model), te svaki entitet mora da poprими te vrijednosti.

**Remove** označi entitet za brisanje iz memorije. Nepoželjno je direktno koristiti operator delete na entitet, zato što postoji šansa da ga neki drugi entitet pozove u sljedećem okviru, što će dovesti do erora.

### 3.2.2.2. Instancijacija entiteta u levelu

Unutar serverske igre, postoji niz od 8192 pokazivača entiteta. Indeksi 0 do 63 su rezervisani za igrače, dok su indeksi 8190 i 8191 rezervisani za „ništa“ entitet i „svijet“ entitet, koji se ne prenose klijentima. Svijet se odnosi na level. Kada se učitava level, instanca klase *GameWorld* čita polja iz tekstualnog zapisa u fajlu levela. Zapis je u sljedećem formatu:

```
{  
    "classname" "func_wall"  
    "origin" "0 0 0"  
    "angles" "0 0 0"  
    "model" "*57"  
    "targetname" "room1_e1_wall"  
}
```

Između vitičastih zagrada nalaze se svojstva entiteta, te svako novo otvaranje i zatvaranje označava novi entitet. Nakon što je level učitao, prolazi se kroz grupe svojstava i čita se polje „classname“. Potom se vrijednost iz tog polja poredi sa listom

registrovanih klasa, te ukoliko je pronađen odgovarajući par, pozove se alokatorska funkcija koja je asocirana sa klasnom registracijom. Ukoliko je broj entiteta prekoračen (8192), engine će odbiti da alocira novi entitet i ostaviti error poruku igraču.

### 3.2.2.3. Registracija entitetskih klasa

Za registraciju entitetskih klasa sa klasnim imenima, napisana je posebna klasa *EntityClassInfo*, skraćeno ECI. Svaka entitetska klasa ima statičnu instancu ove klase. Dio klase prikazan je na slici ispod:

```
class EntityClassInfo
{
public:
    EntityClassInfo( const char* mapClassName, const char* entClassName, c

    // This will be used to allocate instances of each entity class
    // In theory, multiple map classnames can allocate one C++ class
    Entities::IEntity* (*AllocateInstance)();

    // Is this entity of this specific class?
    bool IsClass( const EntityClassInfo& eci ) const
    {
        return classInfoID.GetID() == eci.classInfoID.GetID();
    }

    // Is this entity a subclass of this class?
    bool IsSubclassOf( const EntityClassInfo& eci ) const
    {
        if ( nullptr == super )
            return false;

        if ( classInfoID.GetID() == eci.classInfoID.GetID() )
            return true;

        return super->IsSubclassOf( eci );
    }

    // Get classinfo by map classname
    static EntityClassInfo* GetInfoByMapName( const char* name )
    {
        EntityClassInfo* current = nullptr;
        current = head;

        while ( current )
        {
            if ( !strcmp( current->mapClass, name ) )
                return current;

            current = current->prev;
        }

        return nullptr;
    }
};
```

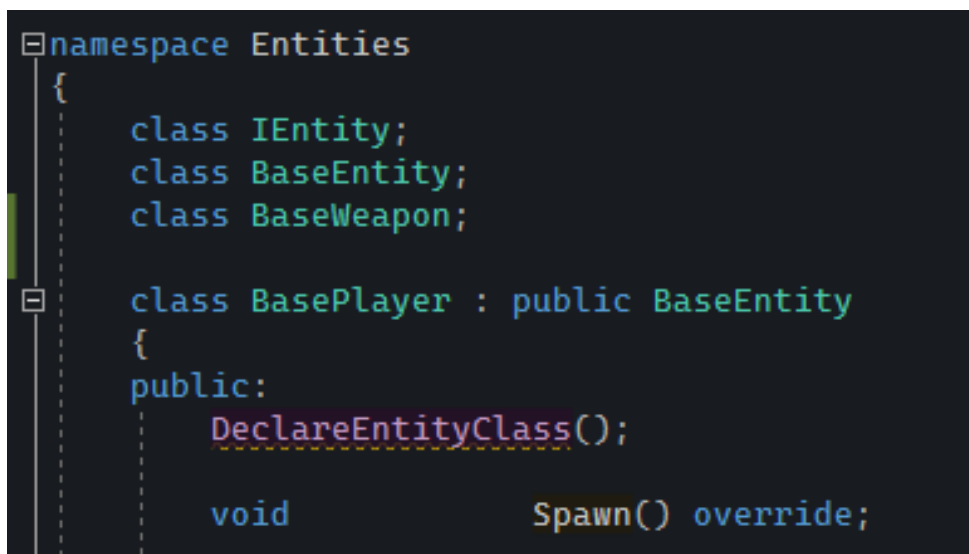
Slika 13. EntityClassInfo klasa

ECI klasa daje mogućnost poređenja nasljedstva i jednakosti klasa. Svaka instanca ECI klase ima svoj jedinstveni ID kojeg automatski određuje kompajler. Taj ID se koristi za poređenje klasa.

Sama registracija klasa odvija se uz pomoć jedan od nekoliko makroa:

- `DeclareEntityClass` – koji se obavezno stavlja u svaku entitetsku klasu
- `DefineEntityClass` – koji se stavlja u implementaciju klase
- `DefineEntityClass_NoMapSpawn` – za registraciju klasa koja se isključivo dinamično alociraju
- `DefineAbstractEntityClass` – za registraciju klasa koje se ne smiju alocirati

Primjer registracije klase dat je na sljedeće dvije slike:

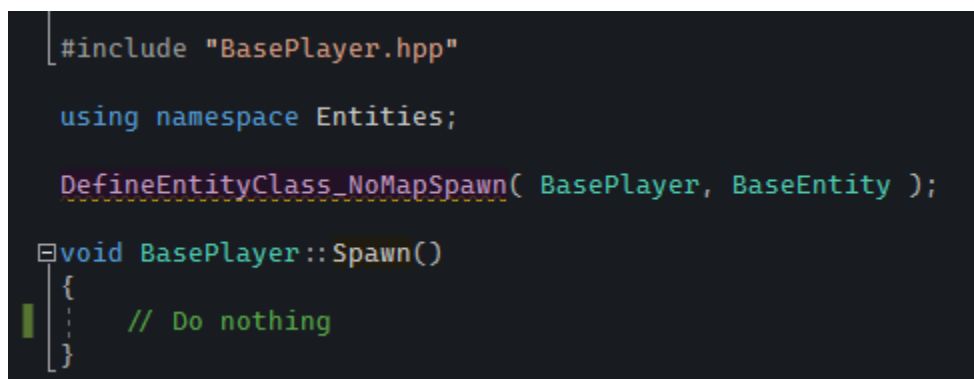


```
namespace Entities
{
    class IEntity;
    class BaseEntity;
    class BaseWeapon;

    class BasePlayer : public BaseEntity
    {
    public:
        DeclareEntityClass();

        void Spawn() override;
    };
};
```

Slika 14. Deklaracija entitetske klase u header fajlu



```
#include "BasePlayer.hpp"

using namespace Entities;

DefineEntityClass_NoMapSpawn( BasePlayer, BaseEntity );

void BasePlayer::Spawn()
{
    // Do nothing
}
```

Slika 15. Definicija entitetske klase u source fajlu

U navedenom primjeru, koristi se registracija za klase koje se alociraju isključivo dinamično. U drugim riječima, instance klase *BasePlayer* se nikad ne mogu instancirati tokom učitavanja levela, nego isključivo poslije.

### 3.2.3. Pregled klasne hijerarhije

Klasna hijerarhija entiteta u igri je sljedeća:

- IEntity (interfejs)
  - BaseEntity (bazna klasa)
    - BaseAI
      - Mercenary
    - BasePlayer
    - BaseTrigger
      - TriggerOnce
      - TriggerMultiple
    - BaseWeapon
      - WeaponFists
      - WeaponPistol
    - BaseMover
      - FuncRotating
      - FuncButton
      - FuncDoor
        - FuncDoorRotating
      - FuncBobbing
      - FuncTrain
    - FuncBreakable
    - FuncToggle
    - FuncStatic
      - FuncDynamic
    - TestModel
    - InfoPlayerStart

Nisu nabrojane sve klase s obzirom na njihovu brojnost.

### 3.3. Implementacija sistema za oružje

Kod akcijskih igara, sistem za oružja je esencijalan. Oružja u akcijskim igrama predstavljaju jednu od veza između igrača i interaktivnih entiteta, kao što su lomljivi predmeti, neprijatelji, eksplozivi i slično.

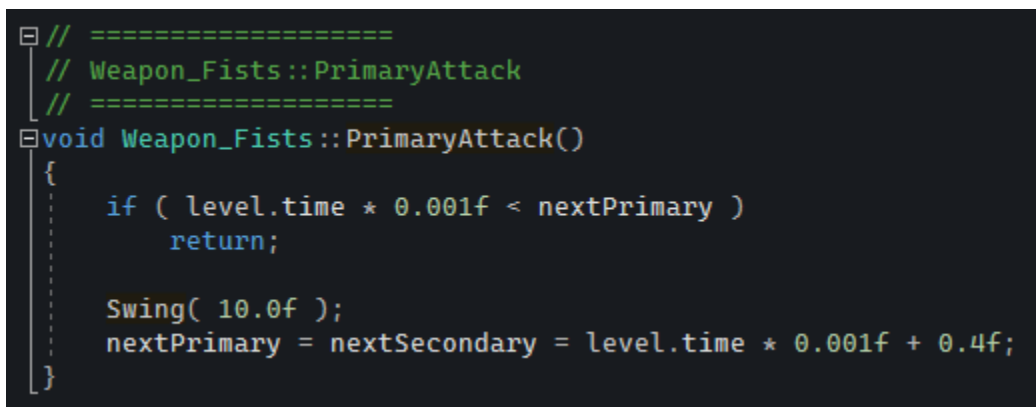
Sistem za oružje, za razliku od entitetskog sistema, zastupljen je i u klijentskoj i u serverskoj igri, dok je entitetski sistem trenutno zastupljen samo u serverskoj igri. Sistem za oružje manifestuje i reguliše korištenje oružja od strane igrača.

#### 3.3.1. Model razdvojenih klasa za klijent i server

Alternativne implementacije objedinjuju klijentsku i serversku igru tako da se isti kod izvrši i na klijentu i na serveru. Međutim, u ovom slučaju, klijent i server izvršavaju oko 80% različitog koda, to jest, kod koji je prisutan na jednoj strani a ne na drugoj i obrnuto.

Sistem za oružje je implementiran na način da postoji entitetska klasa na serverskoj strani, koja implementira logiku oružja, a specijalna klasa na klijentovoj strani, koja implementira estetske aspekte oružja (animacije, manifestacija efekata itd.). Prema tome, ono što se prenosi preko mreže su specijalna stanja igrača. Pogledati sliku 1 za prikaz strukture unutar koje se nalaze spomenuta specijalna stanja.

Primjer implementacije jednog oružja prikazan je na slikama ispod:



```
// =====  
// Weapon_Fists::PrimaryAttack  
// =====  
void Weapon_Fists::PrimaryAttack()  
{  
    if ( level.time * 0.001f < nextPrimary )  
        return;  
  
    Swing( 10.0f );  
    nextPrimary = nextSecondary = level.time * 0.001f + 0.4f;  
}
```

Slika 16. Implementacija primarnog napada na serverskoj strani

Ukoliko je prošlo dovoljno vremena da se primarni napad može ponovno izvršiti, unutar funkcije *Swing* vrši se provjera da li se ispred igrača nalazi ijedan entitet. Ukoliko se nalazi entitet koji je ranjiv (lomljivi predmeti, neprijatelji itd.), primjenjuje se šteta od 10 udarnih bodova. Potom se sljedeći napad odgađa za 0,4 sekunde.

Na sljedećoj slici prikazana je implementacija istog, međutim na klijentskoj strani.

```
void Weapon_Fists::OnPrimaryFire()
{
    if ( cg.time * 0.001f < nextPrimary )
        return;

    if ( Client::IsLocalClient( currentPlayer ) )
    {
        CheckHit();

        renderEntity.StartAnimation( animAttackLeft, true );
        nextPrimary = nextSecondary = cg.time * 0.001f + 0.4f;
        nextIdle = cg.time * 0.001f + renderEntity.GetAnimData( animAttackLeft ).Length();
    }
}
```

Slika 17. Implementacija primarnog napada na klijentskoj strani

Da bi se primarni napad izvršio, potrebno je ispuniti dva uslova. Kao u prethodnom slučaju, potrebno je napad izvršiti u vrijeme koje nije prerano nakon izvršenog napada, međutim, drugi uslov je da izvršilac napada bude lokalni igrač. Pošto server šalje sve vidljive entitete klijentu, a igrači su također entiteti, ostali igrači također izvršavaju interakcije s oružjem, u klijentovom okviru.

To znači da je potrebno razgraničiti koji klijent je lokalni, a koji klijent je sporedni. Zbog toga je napisana pomoćna funkcija *IsLocalClient*.

Ukoliko su oba uslova ispunjena, započinje se animacija na 3D modelu trenutnog oružja, te se uspostavlja vrijeme za sljedeći napad, koji obavezno mora biti jednak onome na serveru. Varijabla *nextIdle* određuje kada će se sljedeća animacija reproducirati, a to je trenutno vrijeme + dužina trenutne animacije.

Razdvojeni model ima prednost da klijent izvršava samo onaj kod koji je njemu relevantan, a isto vrijedi i za servera. Međutim, mana razdvojenog modela je veća mogućnost nesaglasnosti između klijenta i servera, povećana težina debugovanja kao i povećana težina pisanja koda.

### 3.3.2. Interpretiranje interakcija s oružjem

U strukturi *usercmd\_t* nalazi se varijabla *interactionButtons*, tipa 16-bitni integer. Svaki bit se koristi za jedinstveni događaj u vezi igračeve interakcije, nezavisno od konteksta (interakcija s oružjem, okolinom, likovima itd.). Trenutno, postoje sljedeće vrste interakcije:

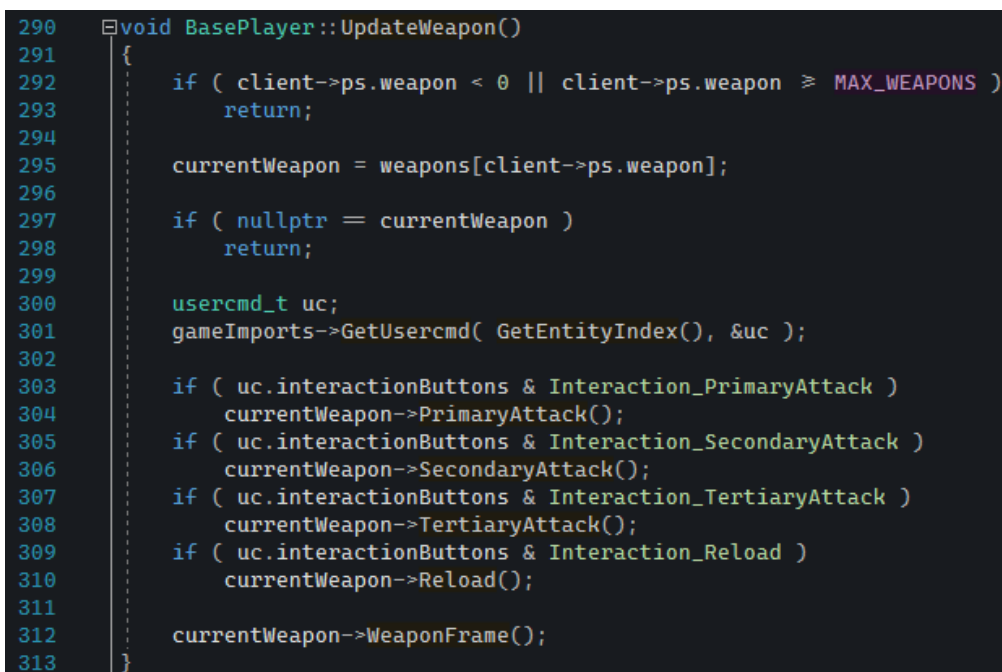
- Primarni napad, bit 0
- Sekundarni napad, bit 1
- Tercijarni napad, bit 2
- Punjenje, bit 3



- Korištenje predmeta, bit 4
- Korištenje predmeta u inventaru, bit 5

Iskorišteno je prvih 6 bitova, dok se ostalih 10 može iskoristiti u budućnosti. Premda su primarni, sekundarni i tercijarni napad označeni kao „napadi“, ne moraju isključivo biti korišteni za napadanje. Ukoliko igrač nosi alat poput voltmetra, primarni napad bi vršio mjerenje tim alatom, dok bi sekundarni napad vršio mijenjanje domene mjerenja.

Na serverskoj strani, svaki igrač provjerava zastavice unutar varijable *interactionButtons* i zavisno od njih, poziva određene funkcije na igračevom trenutno izabranom oružju. Pogledati sliku ispod.



```
290 void BasePlayer::UpdateWeapon()
291 {
292     if ( client->ps.weapon < 0 || client->ps.weapon ≥ MAX_WEAPONS )
293         return;
294
295     currentWeapon = weapons[client->ps.weapon];
296
297     if ( nullptr == currentWeapon )
298         return;
299
300     usercmd_t uc;
301     gameImports->GetUsercmd( GetEntityIndex(), &uc );
302
303     if ( uc.interactionButtons & Interaction_PrimaryAttack )
304         currentWeapon->PrimaryAttack();
305     if ( uc.interactionButtons & Interaction_SecondaryAttack )
306         currentWeapon->SecondaryAttack();
307     if ( uc.interactionButtons & Interaction_TertiaryAttack )
308         currentWeapon->TertiaryAttack();
309     if ( uc.interactionButtons & Interaction_Reload )
310         currentWeapon->Reload();
311
312     currentWeapon->WeaponFrame();
313 }
```

Slika 18. Interakcija s oružjem na serverskoj strani

Prije nego što se funkcije oružja pozovu, potrebno je provjeriti da li trenutno oružje postoji. Ukoliko je trenutno oružje nul-pokazivač, rukovanje interakcija se preskače.

Postoje dva jednostavna mehanizma za otkrivanje da li trenutno oružje postoji. Prvi je provjeravanje da li je nul-pokazivač. Drugi mehanizam je provjera indeksa oružja. Indeks oružja se nalazi u strukturi stanja igrača, odnosno, to je varijabla *weapon*. Ukoliko je *weapon* manje od 0 ili veće od konstante *MAX\_WEAPONS* (32), indeks je nevažeći i interakcija se preskače.

Sličan proces se dešava na klijentskoj strani, pogledati sliku ispod.

```

30 // =====
31 // Client::Update
32 // =====
33 void Client::Update()
34 {
35     if ( vegetationSystem )
36     {
37         vegetationSystem->Update();
38         vegetationSystem->Render();
39     }
40
41     usercmd_t uc = GetUsercmd();
42
43     auto weapon = GetCurrentWeapon();
44
45     if ( weapon )
46     {
47         if ( uc.interactionButtons & Interaction_PrimaryAttack )
48             weapon->OnPrimaryFire();
49
50         if ( uc.interactionButtons & Interaction_SecondaryAttack )
51             weapon->OnSecondaryFire();
52
53         if ( uc.interactionButtons & Interaction_TertiaryAttack )
54             weapon->OnTertiaryFire();
55
56         if ( uc.interactionButtons & Interaction_Reload )
57             weapon->OnReload();
58     }
59
60     // If the game is paused, pause the music too
61     trap_DM_Pause( false, !IsPaused() );
62 }

```

Slika 19. Interakcija s oružjem na klijentskoj strani

Kada bi kod za oružje iz navedene funkcije bio izoliran, izgledalo bi da je procedura jednostavnija kod klijenta nego kod servera. Međutim, kompleksnost je jednaka, kad pogledamo definiciju funkcije *GetCurrentWeapon*, slika ispod:

```

// =====
// Client::GetCurrentWeapon
// =====
ClientEntities::BaseClientWeapon* Client::GetCurrentWeapon()
{
    if ( cg.snap->ps.weapon < 0 || cg.snap->ps.weapon ≥ MAX_WEAPONS )
        return nullptr;

    ClientEntities::BaseClientWeapon* weapon = gWeapons[cg.snap->ps.weapon];
    if ( weapon )
        return weapon;

    return nullptr;
}

```

Slika 20. Definicija funkcije *GetCurrentWeapon*, koja vraća pokazivač na trenutno oružje

### 3.3.3. Vršenje napada

Da bi oružja bila funkcionalna, obavezna su činiti znatnu količinu štete zahvaćenim entitetima. Logika napadanja se implementira na serverskoj strani. Moguće je istu implementirati na klijentskoj strani, no isključivo za estetske potrebe. Ukoliko se „serverski“ napad vrši na klijentskoj strani, to će vidjeti samo klijent, a ostali neće, stvarajući iluziju da je igrač pogodio nešto, što nije slučaj.

Premda svaka klasa oružja može implementirati tri različite funkcije za napad, ne znači da oružja ne mogu napadati na više od tri načina. U drugim riječima, naprimjer, unutar funkcije za primarni napad, može se implementirati jedna vrsta napada, a može se implementirati i više.

Najjednostavniji oblik napadanja se vrši na sljedeći način, u koracima:

1. Izračunati pravac  $V_{\text{forward}}$  u kojem igrač gleda, zavisno od uglova gledanja.
2. Izvesti traganje zrake (eng. *ray trace*)<sup>6</sup> počevši od glave igrača, u pravcu  $V_{\text{forward}}$ , dužine  $X$  unita, pohraniti rezultate traganja u privremenu varijablu.
3. Ukoliko se među rezultatima nalazi indeks entiteta (zraka je pogodila entitet), traži se pokazivač na entitet, i poziva se njegova *TakeDamage* funkcija.

Ova tehnika se naziva skeniranje pogotka (eng. *hit scan*), i idealna je za većinu vatrenog oružja. S obzirom da se meci u stvarnosti kreću jako visokom brzinom, ima smisla koristiti skeniranje pogotka ako igra zauzima mjesto u prostorijama i koridorima. Međutim, skeniranje pogotka nije pogodna opcija za velike udaljenosti, kao što je pucanje u metu koja je udaljena preko 1500 m.

Alternativna implementacija, koja je pogodna za veće udaljenosti, se oslanja na slanje projektila visoke brzine u koraku 2, dok je korak 3 provjeravanje dodira projektila sa bilo kojim entitetom. Nedostatak takve implementacije leži u činjenici da projektili koriste memoriju, jer se moraju alocirati, a prednost je veća fizička tačnost, i mogućnost simulacije raznih okolišnih parametara (npr. opadanje na daljinu, utjecaj vjetra itd.).

U određenim vrstama oružja, projektili su idealni, naprimjer kod bacača raketa, ručnih bombi i slično.

Pored navedene dvije vrste vršenja napada, postoji napadanje po radijusu, gdje je zahvaćen bilo koji entitet koji je unutar dometa određene tačke. U slučaju projektila rakete i bombe, prilikom dodira sa bilo kojim entitetom, izvršavaju napadanje po radijusu, a količina štete opada linearno sa udaljenošću od tačke napada.

---

<sup>6</sup> „How do bullets work in video games?“, Tristan Jung, jul 2018. - <https://medium.com/@3stan/how-do-bullets-work-in-video-games-d153f1e496a8>

## 3.4. Igrač

Igrač je osnovni vršilac radnje u igri, u pojedinim slučajevima i pokretač. U singleplayer igrama, postoji samo jedan igrač, dok u multiplayer igrama konkurentno učestvuju više igrača.

Igra daje brojne mehanizme igraču sa kojima može upravljati jednim likom ili sa više likova unutar igre. U slučaju akcijske igre iz prvog lica, upravlja se samo jednim likom, a svijet je igraču predstavljen iz perspektive prvog lica. Takvi mehanizmi odnose se na kontrole koje su mapirane za tipke na tastaturi, mišu ili kontroleru (primjer: tipka W za kretanje naprijed, tipka S za kretanje nazad).

### 3.4.1. Implementacija igrača na klijentu

Na klijentskoj strani, pojmovi „igrač“ i „klijent“ se skoro poistovjećuju. Igrač na klijentskoj strani popunjuje varijable iz strukture *usercmd\_t* (slika 1.), a oslanja se na game engine da otkrije kada igrač pritišće određenu tipku.

Svaka komanda ima svoj naziv, a neke posjeduju i asocirani bit interakcije, kao što je evidentno na slici ispod:

```
BaseButtonFunction buttonFunctions[] =
{
    // movement buttons
    SingleButtonFunction<&in_forward>( "forward" ),
    SingleButtonFunction<&in_back>( "back" ),
    SingleButtonFunction<&in_speed>( "speed" ),
    SingleButtonFunction<&in_moveleft>( "moveleft" ),
    SingleButtonFunction<&in_moveright>( "moveright" ),

    // jumping and crouching
    SingleButtonFunction<&in_up>( "moveup" ),
    SingleButtonFunction<&in_down>( "movedown" ),

    // turning
    SingleButtonFunction<&in_left>( "left" ),
    SingleButtonFunction<&in_right>( "right" ),

    // misc
    SingleButtonFunction<&in_strafe>( "strafe" ),
    SingleButtonFunction<&in_lookup>( "lookup" ),
    SingleButtonFunction<&in_lookdown>( "lookdown" ),

    // generic buttons
    ButtonFunction<Button_Attack, in_buttons>( "attack" ), // Deprecated command, don't use
    ButtonFunction<Button_Talk, in_buttons>( "talk" ),

    // interaction buttons
    ButtonFunction<Interaction_PrimaryAttack, in_interactionButtons>( "attack1" ),
    ButtonFunction<Interaction_SecondaryAttack, in_interactionButtons>( "attack2" ),
    ButtonFunction<Interaction_TertiaryAttack, in_interactionButtons>( "attack3" ),
    ButtonFunction<Interaction_Reload, in_interactionButtons>( "reload" ),
    ButtonFunction<Interaction_Use, in_interactionButtons>( "use" ),
    ButtonFunction<Interaction_UseItem, in_interactionButtons>( "useitem" ),
};
```

Slika 21. Registracija korisničkih komandi za unos

Na taj način mapiraju se komande za unosne uređaje. Varijable čiji je naziv prefiksiran sa „in\_“ su instance strukture *kbutton\_t*, koja je prikazana na slici ispod:

```
//
// cl_input
//
typedef struct {
    int         down[2];           // key nums holding it down
    unsigned    downtime;         // msec timestamp
    unsigned    msec;             // msec down this frame if both a down and up happened
    qboolean    active;           // current state
    qboolean    wasPressed;       // set when down, not cleared when up
} kbutton_t;
```

Slika 22. Struktura *kbutton\_t*

Engine će asinhrono provjeravati da li korisnik pritisće zadanu tipku, i bilježit će koliko dugo je ista zadržana.

Komande koje su asocirane s interakcionim bitom će da mijenjaju varijablu *interactionButtons* u strukturi *usercmd\_t*, a iste će kasnije proslijediti serveru.

Drugi dio koda na klijentskoj strani, koji se odnosi na igrača, je klasa *ClientView*. Premda se koristi naziv View, nema sličnosti sa Model-View-Controller uzorkom. Pojam „view“ se odnosi na pogled, odnosno vid igrača. Pregled klase dat je na slici ispod:

```
class ClientView final
{
    struct ViewShake final { ... };
    struct ViewPunch final { ... };

    constexpr static size_t MaxViewShakes = 32U;

public:
    ClientView();

    // Calculating the view stuff includes view bobbing etc.
    void CalculateViewTransform( Vector& outOrigin, Vector& outAngles );
    // Calculating weapon stuff includes weapon bobbing, swaying etc.
    void CalculateWeaponTransform( Vector& outOrigin, Vector& outAngles );
    // Adds a view shake - every time a quake happens, explosions etc.
    void AddShake( float frequency, float duration, Vector direction );
    // Gets the average of all shakes
    Vector CalculateShakeAverage() const;
    // Adds a view punch - every time a weapon is fired etc.
    void AddPunch( float duration, Vector angles );
    // Gets the total of all view punches
    Vector CalculatePunchTotal() const;

    const Vector& GetViewOrigin() const { return currentViewOrigin; }
    const Vector& GetViewAngles() const { return currentViewAngles; }
    const Vector& GetWeaponOrigin() const { return currentWeaponOrigin; }
    const Vector& GetWeaponAngles() const { return currentWeaponAngles; }
```

Slika 23. *ClientView* klasa

Unutar *ClientView* klase, nalaze se klase *ViewShake* i *ViewPunch*, koje su zaslužne za efekte tresanja i udaranja ekrana. Funkcija *CalculateViewTransform* izračunava poziciju igračevog vida kao i njegovu rotaciju, dok *CalculateWeaponTransform* čini isto, samo za igračevo oružje. Funkcije ispod su vezane za vidne efekte.

Pomoću analogije, igračev vid moguće je zamisliti poput nevidljive kamere koja lebdi u zraku. Pogledati sliku ispod:



Slika 24. Vid igrača 1

Na desnoj strani slike prikazano je oružje, odnosno njegov 3D model, dok je na lijevoj strani drugi igrač (igrač 2). Igrač 1 ne vidi korisnički interfejs od igrača 2, niti njegovo oružje na način kako je njemu predstavljeno. U drugim riječima, oba igrača vide iz različitih perspektiva.

### 3.4.2. Implementacija igrača na serveru

Na serverskoj strani, igrač je implementiran u klasi *BasePlayer*. Unutar nje, nalazi se *TakeDamage* funkcija da bi igrač mogao primiti štetu od napada, pokazivači na entitete oružja, funkcije za teleportaciju i slično. Prototip klase *BasePlayer* prikazan je na slici ispod:

```
class BasePlayer : public BaseEntity
{
public:
    DeclareEntityClass();

    void        Spawn() override;

    void        TakeDamage( IEntity* inflictor, IEntity* attacker, int damageFlags, float damageDealt ) override;

public: // "Properties"
    gclient_t*   GetClient();
    void        SetClient( const gclient_t* playerClient );

    Vector       GetClientViewAngle() const;
    void        SetClientViewAngle( const Vector& newAngle );

public: // Weapons
    void        AddWeapon( BaseWeapon* weapon );
    BaseWeapon* GetCurrentWeapon();
    bool        HasAnyWeapon();
    bool        HasWeapon( int weaponID );
    void        SendWeaponEvent( uint32_t weaponEvent );
    void        UpdateWeapon();

protected:
    BaseWeapon*  currentWeapon{ nullptr }; // the currently selected weapon
    BaseWeapon*  weapons[MAX_WEAPONS]; // weapon inventory

public: // Misc
    void        ClientCommand();
    // Uses the object the player is looking at
    void        PlayerUse();
    // Get the position of the player's "eyes"
    Vector       GetViewOrigin();
    // Aka spawn a dead body
    void        CopyToBodyQue();
    // Add an event to be played back by the client
    void        AddEvent( int event, int eventParameter ) override;
    // Forcibly fire a weapon
    void        FireWeapon();
    void        SetTeam( const char* teamName );
    void        StopFollowing();
    void        FollowCycle( int dir );
    // Burn from lava, drowning etc.
    void        WorldEffects();
    // Applies all the damage taken this frame
    void        ApplyDamage();
    // Teleports the player to a given place
    void        Teleport( const Vector& toOrigin, const Vector& toAngles );
};
```

Slika 25. Klasa *BasePlayer*

S obzirom da je *BasePlayer* bazna klasa, programer igre može napisati klase koje nasljeđuju od *BasePlayer*. Na taj način se postižu različite varijacije i uloge igrača, sa različitim atributima.

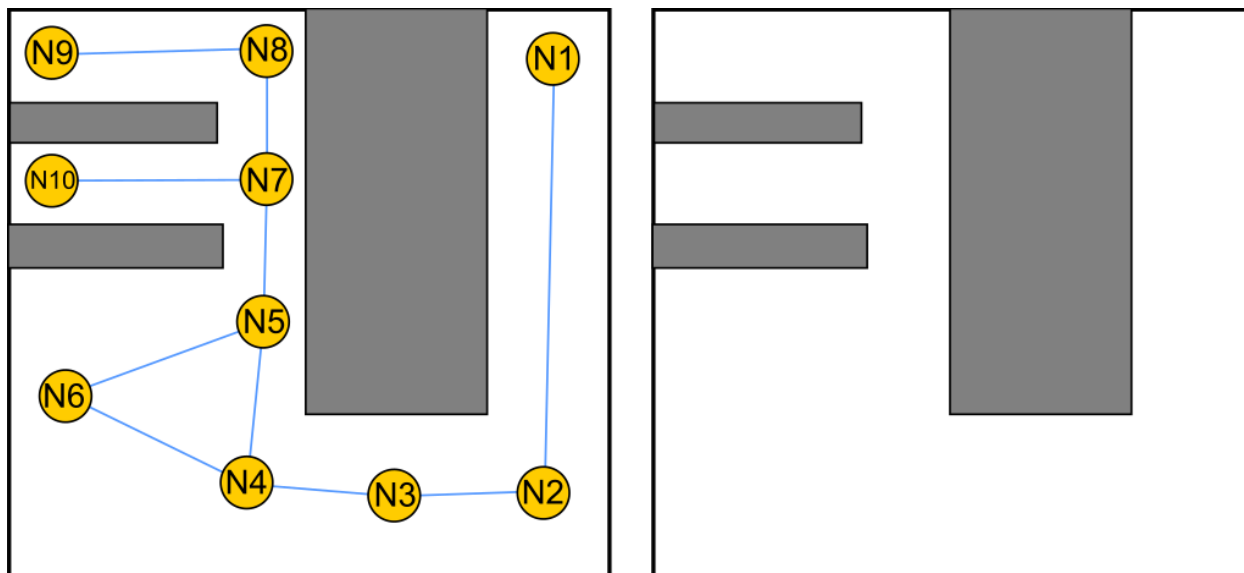


## 3.5. AI sistem

AI sistem, odnosno sistem umjetne inteligencije, služi za logičku manifestaciju likova u igri, kojima nijedan igrač ne upravlja direktno (NPC, eng. *non-player character*, neigrački karakter). NPC-ovi mogu služiti funkciju neprijateljskih likova u igri, kao i prijateljskih i neutralnih, međutim, nisu ograničeni na takve. NPC-ovi mogu biti životinje, roboti, vozila i slično.

### 3.5.1. Snalaženje u prostoru

Snalaženje u prostoru je odgovor na pitanje kako doći od tačke A do tačke B. Proces započinje tako što level dizajner stavlja tačkaste entitete u level, takozvane „node“, koje on/ona povezuje. Ukoliko jedna noda ima više od dvije veze, za tu nodu kažemo da je čvor. Ilustracija jednog levela i noda za taj level data je na slici ispod:



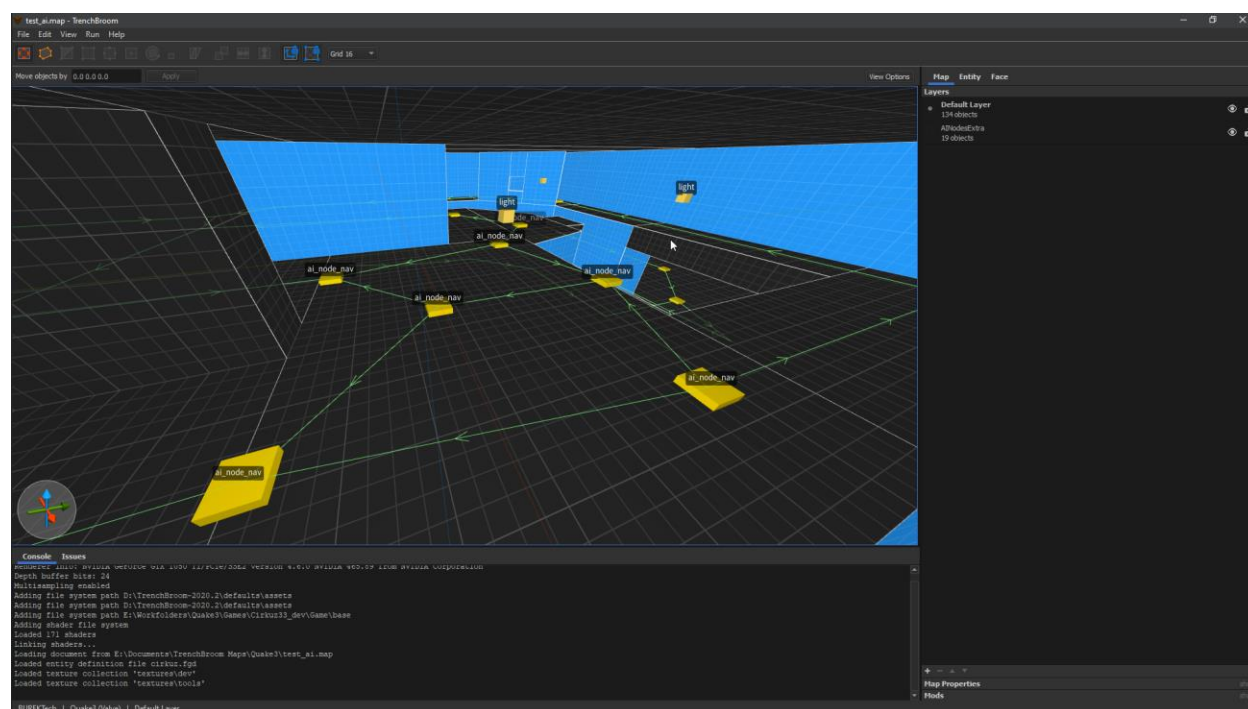
Slika 26. Level sa 3 zida (desno, zidovi označeni sivom bojom), i povezanim nodama (lijevo)

Plave linije između pojedinih noda su putevi, dok je, naprimjer, noda N5 čvorna noda, pošto ima tri veze. Skup puteva između noda N1 i N4 naziva se putanja, a node N1, N9 i N10 su krajevi. U drugim riječima, skupovi puteva između čvorova i/ili krajeva nazivaju se putanje.

U kodu, postoji *Node* klasa koja sadrži algoritme za traženje noda zavisno od željene pozicije kretanja. Potom se vraća sljedeća noda koju je potrebno preći da bi se došlo do željene destinacije.

Level dizajner stavlja node u level i veže iste, primjer čega je prikazan na slici ispod:





Slika 27. Level editor u levelu sa povezanim entitetima klase `ai_node_nav`

Važno je napomenuti da su takvi entiteti nevidljivi u igri. Oni su isključivo logički entiteti.

## 3.5.2. Pamćenje

Svaki NPC ima sposobnost pamćenja događaja, gdje može pamtiiti gdje je vidio strani entitet, gdje je čuo strani entitet, kao i razlikovanje između različitih vrsta sjećanja (dobro sjećanje, loše sjećanje usljed pucnjave itd.).

Implementacija memorije počinje sa klasom *MemoryFrame* koja predstavlja jedan memorijski okvir i sadrži najosnovnije podatke. (pogledati sliku ispod)

```
class MemoryFrame final
{
public:
    void Reset();

    Vector lastSeen{ Vector::Zero }; // where did I last see the entity?
    Vector lastHeard{ Vector::Zero }; // where did I last hear the entity?

    uint32_t flags{ 0U }; // various memory flags
    float time{ -1.0f }; // when did I get this information?
};
```

Slika 28. Klasa *MemoryFrame*

Međutim, memorijski okviri sami od sebe nisu iskoristivi za identifikaciju saveznika, neprijatelja i sličnog. Memorijski okviri se manifestuju u klasi *EntityMemory*, koja sadrži indeks entiteta koji se pamti, i fiksirani niz od 16 memorijskih okvira.

```
class EntityMemory final
{
public:
    constexpr static size_t MaxMemoryFrames = 16U;
    constexpr static float AwarenessExpiration[AI::Awareness_MAX] =
    {
        100.0f, // None
        3.0f, // Imagining
        15.0f, // Suspicious
        60.0f, // Certain
        240.0f, // MaximumCertainty
    };

    const static EntityMemory NoMemory;

    EntityMemory();
    EntityMemory( BaseEntity* ent, AI::Relationship relationship = AI::Relationship::None );

    void          AddFrame( const MemoryFrame& frame );
    void          Update( const float& time );

    Vector        LastSeen() const;
    Vector        LastHeard() const;
    float         LastMemoryTime() const;

    void          IncreaseAwareness( const float& time );
    void          DecreaseAwareness();

    bool          alive{ true };
    uint8_t       awareness{ AI::Awareness::None };
    float         lastAware{ 0.0f };
    AI::Relationship relationship;

    uint16_t       entityIndex{ ENTITYNUM_NONE };
    MemoryFrame    frames[MaxMemoryFrames];
};
```

Slika 29. Klasa *EntityMemory*

Broj maksimalnih memorijskih okvira se može proizvoljno mijenjati, s tim da je potrebno paziti da se ne troši previše memorije.

Dok je NPC živ, konstantno će održavati listu *EntityMemory* objekata, te ukoliko „zapamti“ veliki broj entiteta, može doći do prevelike upotrebe računarske memorije. U konfiguraciji sa 16 memorijskih okvira, jedan *EntityMemory* objekat zauzima 528 bajtova, tako da, ukoliko jedan NPC zapamti 500 entiteta, konzumirat će aproksimativno 257 KB.

S obzirom da u jednom levelu može biti više NPC-ova, naprimjer, u urbanoj sredini može biti preko 200 istih, moguće je iskoristiti 50 MB memorije. Međutim, u praksi se sigurno neće desiti da jedan NPC pamti više od 50 entiteta.

### 3.5.3. Detekcija neprijatelja

U definiciji klase *EntityMemory*, nalazi se varijabla *awareness*, u prevodu svijest. NPC će periodično provjeravati svoje vidno polje i održavati listu entiteta koje trenutno vidi. Potom će, iz te liste, provjeravati koji su entiteti neprijatelji i obrnuto. Ovo je implementirano u funkciji *SightQuery*, slika ispod:

```
void Mercenary::SightQuery()
{
    auto entities = gameWorld->EntitiesInRadius( GetCurrentOrigin(), 1024 );

    Vector forward;
    Vector::AngleVectors( viewAngles, &forward, nullptr, nullptr );

    for ( IEntity*& ent : entities )
    {
        // Step 0: see if it's a character of any kind
        if ( !ent->IsSubclassOf( Mercenary::ClassInfo ) && !ent->IsClass( BasePlayer::ClassInfo ) )
            continue;

        // Step 1: determine if in FOV
        Vector dir = (ent->GetCurrentOrigin() - GetCurrentOrigin()).Normalized();
        float dot = forward * dir;

        // Hardcoded 120-degree FOV, todo: make more customisable
        if ( dot < 0.5f ) // Not visible
            continue;

        // Step 2: determine if visible, trace from head to head, todo: trace several other bodyparts too
        Vector start = GetCurrentOrigin() + GetHeadOffset();
        Vector end = ent->GetAverageOrigin() + ent->GetCurrentOrigin();
        trace_t tr;

        Util::Trace( &tr, start, nullptr, nullptr, end, GetEntityIndex(), MASK_SHOT );
        if ( tr.entityNum != ent->GetEntityIndex() ) // Not visible
            continue;

        // Step 3: determine if this is a friend or foe, "remember" them if possible
        RegisterVisibleEntity( static_cast<BaseEntity*>( ent ) );
    }
}
```

Slika 30. Implementacija funkcije SightQuery

Na početku funkcije, dobiva se lista entiteta u dometu od 1024 unita. Potom se provjerava da li je entitet karakter. Trenutno postoje dvije vrste karaktera, a to su neprijateljski čuvar i igrač. Ukoliko nije karakter, preskače se.

Potom se provjerava da li je entitet u vidnom polju trenutnog NPC-a. Ukoliko nije, preskače se, odnosno NPC ga ne vidi. Na kraju, testira se zraka od očiju NPC-a do centra entiteta. Ukoliko ništa nije prekinulo zraku, entitet je zaista vidljiv, i registrira se kao takav, da bi NPC mogao da ga zapamti i vrši ostalo obrađivanje.

Sljedeći korak je povećati varijablu *awareness* iz *EntityMemory* objekta. To se radi unutar *RegisterVisibleEntity* funkcije koja poziva *IncreaseAwareness*, slika ispod:

```
void Mercenary::RegisterVisibleEntity( BaseEntity* ent )
{
    AI::Relationship rel = Relationship( ent );

    // If this entity is not in memory, remember it
    if ( !IsInMemory( ent ) )
    {
        memories.push_back( EntityMemory( ent, rel ) );
    }

    // Get some data about the entity to remember for a while
    MemoryFrame mf;
    EntityMemory* em = GetMemory( ent );

    // Since RegisterVisibleEntity gets called from SightQuery,
    // it only makes sense to set lastSeen
    mf.lastSeen = ent->GetCurrentOrigin();
    mf.time = level.time * 0.001f;

    em->AddFrame( mf );
    em->Update( level.time * 0.001f );
    em->IncreaseAwareness( level.time * 0.001f );
}
```

Slika 31. Implementacija funkcije *RegisterVisibleEntity*

Ukoliko se NPC „ne sjeća“ entiteta (nije u memoriji), on se prvo dodaje u memoriju. Potom se popunjava novi memorijski okvir sa informacijom gdje i kada je entitet primijećen. Potom se okvir dodaje u memoriju i povećava se svijest o istom.

Važno je napomenuti da NPC ima sposobnost da zaboravi na entitet ukoliko je prošlo dovoljno dugo vremena.

#### 3.5.4. Evaluacija situacije

Evaluacija situacije je mehanizam za procjenu sljedeće akcije NPC-a. Pomoću evaluacije, NPC može zaključiti koju će sljedeću akciju uraditi, npr. trčati na određeno mjesto, sakrivati se, napadati, pritisnuti tipku itd.

Svaki NPC ima svoj doživljaj situacije (varijabla *situation*), a trenutno može biti jedan od sljedećih: opušteni, borbeni, pobjednički, porazni. Evaluacija i promjena situacije je implementirana kao switch case iskaz, koji poziva različite funkcije zavisno od trenutne situacije.

Procjena situacije se vrši u funkciji *EvaluateSituation*, kao na slici ispod:

```
689 void Mercenary::EvaluateSituation()
690 {
691     switch ( situation )
692     {
693         case AI::ST_Casual: Situation_Casual(); break;
694         case AI::ST_Combat: Situation_Combat(); break;
695     }
696
697     UpdatePath();
698 }
699
700 void Mercenary::Situation_Casual()
701 {
702     // Check if there are any enemies so we can transition to combat mode
703     BaseEntity* enemy = GetEnemy();
704
705     if ( enemy )
706     {
707         situation = AI::ST_Combat;
708         targetEntity = enemy;
709         return;
710     }
711 }
712
713 void Mercenary::Situation_Combat()
714 {
715     moveIdeal = targetEntity->GetCurrentOrigin();
716     lookTarget = targetEntity->GetCurrentOrigin();
717 }
```

Slika 32. Implementacija funkcija *EvaluateSituation*, i različitih situacionih funkcija

U funkciji *Situation\_Casual*, koja djeluje kada je doživljaj situacije opušten, postoji provjera za neprijateljem. Ukoliko NPC zna da postoji neprijatelj u blizini, prebacit će doživljaj situacije sa opuštenog na borbeni, te će se ubuduće izvršavati funkcija *Situation\_Combat*.

Ovaj algoritam se može dalje proširiti, da NPC obavlja određeni posao ukoliko je doživljaj situacije opušten, ili da brani određen objekat ukoliko je u borbenom stanju.

### 3.5.5. Klasifikacija prijatelja i neprijatelja

Unutar klase *EntityMemory*, nalazi se atribut *relationship*, tipa *AI::Relationship*. U prevodu znači odnos, a odgovoran je za uspostavljanje odnosa između NPC-ova i igrača. Tip *AI::Relationship* je enumeracija sljedećih simbola: ništa, najbolji saveznik, saveznik, neutralan, neprijatelj i najgori neprijatelj.

Određivanje odnosa se vrši u NPC-ovoj funkciji *Relationship*, koja za parametre prima vrstu (*species*) i stranu (*faction*), potom se definiše tabela odnosa strana, i primjenjuje se poseban slučaj za vrste. Implementacija takve funkcije data je na slici ispod:

```
AI::Relationship Mercenary::Relationship( uint8_t spec, uint8_t fac ) const
{
    // to save some typing
    using R = AI::Relationship;

    constexpr static AI::Relationship table[Faction_MAX][Faction_MAX] =
    {
        // Faction:      None      Mafia      Criminals  Police      Agency      Aliens      Rogue
        /*None*/      { R::Neutral, R::Neutral, R::Neutral, R::Neutral, R::Neutral, R::Enemy, R::Enemy },
        /*Mafia*/      { R::Neutral, R::BestAlly, R::Enemy, R::Enemy, R::WorstEnemy, R::Enemy, R::Enemy },
        /*Criminals*/ { R::Neutral, R::Enemy, R::Ally, R::Enemy, R::WorstEnemy, R::Enemy, R::Enemy },
        /*Police*/     { R::Neutral, R::Enemy, R::Enemy, R::Ally, R::Enemy, R::Enemy, R::Enemy },
        /*Agency*/    { R::Neutral, R::WorstEnemy, R::Enemy, R::Enemy, R::Ally, R::Enemy, R::Enemy },
        /*Aliens*/     { R::Enemy, R::Enemy, R::Enemy, R::Enemy, R::Enemy, R::Ally, R::Enemy },
        /*Rogue*/      { R::Enemy, R::Enemy, R::Enemy, R::Enemy, R::Enemy, R::Enemy, R::Enemy },
    }; //Species^

    switch ( spec )
    {
        case Species_Object:
        case Species_Plant:
            return R::None;
    }

    return table[faction][fac];
}
```

Slika 33. Implementacija funkcije Relationship

Međutim, pošto igrač nema atribut vrste i strane, za njega vrijedi poseban slučaj, kao što je prikazano na slici ispod:

```
AI::Relationship Mercenary::Relationship( BaseEntity* entity ) const
{
    if ( entity->IsSubclassOf( Mercenary::ClassInfo ) )
    {
        Mercenary* merc = static_cast<Mercenary*>(entity);
        return Relationship( merc->GetSpecies(), merc->GetFaction() );
    }
    else if ( entity->IsClass( BasePlayer::ClassInfo ) )
    {
        return Relationship( Species_Human, Faction_Agency );
    }
    else
    {
        return AI::Relationship::None;
    }
}
```

Slika 34. Implementacija funkcije Relationship koja za parametar prima entitet

Dakle, ukoliko je entitet koji se prosljeđuje igrač, za njega uvijek vrijedi jedna vrsta (*Species\_Human*) i jedna strana (*Faction\_Agency*). Ostali NPC-ovi se porede prema svojim atributima, koje može podesiti level dizajner da bi stvorio borbene situacije bez učešća igrača.

### **3.6. Pravila igre i brojnost igrača**

Pravila igre se mogu implementirati sa specijalnom klasom, no, mogu se implementirati samom funkcionalnošću različitih entitetskih klasa. Pravila igre definišu ono što igrač može, odnosno ne može raditi unutar igre.

Pored pravila igre, postoji i koncept brojnosti igrača, koji označava koliko igrača učestvuje u igri.

#### **3.6.1. Singleplayer**

Singleplayer način igranja je takav način igranja gdje samo jedan igrač izvršava zadatke da bi prešao level. Najčešće igrač međudjeluje sa NPC-ovima koji mogu biti prijateljski ili neprijateljski nastrojeni.

#### **3.6.2. Multiplayer**

Multiplayer način igranja je takav način igranja gdje učestvuje više igrača, najčešće jedan protiv drugog. NPC-ovi nisu prisutni u najviše slučajeva.

Multiplayer igre (mečevi) se najčešće dešavaju na jednom levelu koji je specijalno dizajniran za takav način igranja, a zadatak je dobiti više bodova od ostalih igrača na način koji je definisan pravilima igre.

U *BUREKTech* engine-u, maksimalni teoretski broj igrača je 256, a uobičajena vrijednost je 64.

#### **3.6.3. Kooperacija**

Kooperacija ili saradnja je poseban slučaj multiplayer načina igranja, gdje više igrača sarađuje da bi zajedno prešli level. NPC-ovi su često prisutni, najčešće kao neprijatelji kojima je cilj spriječiti igrače od napredovanja.

## 4. Sadržaj i dizajn igre

### 4.1. Osnovni podaci

Igre, između ostalog, definiše i njihov žanr. Postoje različiti žanrovi, od kojih su najosnovniji: avanturistički, akcijski, zagonetni (puzzle), trkaći itd.

Igra čiju izradu dokumentira Rad je akcijskog žanra, preciznije akcijska pucačina iz prvog lica.

Igra sadrži osnovne mehanike kao što su kretanje, trčanje, skakanje, čučanje i gledanje. Pored toga, igraču je omogućena interakcija s okruženjem na sljedeće načine: korištenje tipki, poluga, vrata, kapija, napdanje lomljivih objekata u svrhe oslobađanja puta, otključavanje vrata i više.

Glavni izbornik igre dat je na slici ispod:



Slika 35. Glavni izbornik (main menu) igre



Grafike igre, odnosno vizuelni dizajn i stil igre, pripada tzv. „retro“ pokretu<sup>7</sup>, kojem je cilj da oponaša igre koje se smatraju vrlo starim. Izgled igre prikazan je na slici ispod:



Slika 36. Level u šumi

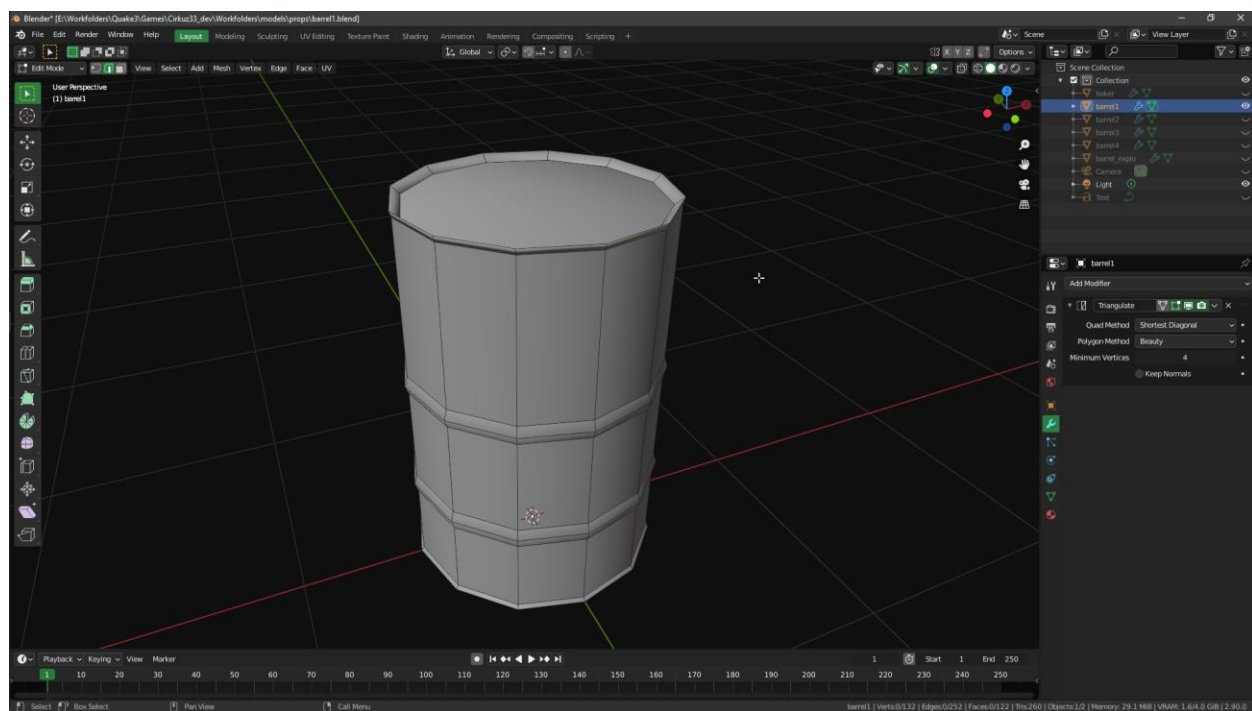
## 4.2. Izrada 3D modela

Svi 3D modeli igre stvoreni su u programu *Blender*<sup>8</sup>. Izrada modela započinje sa osnovnim oblikom modela (slika 37), nakon čega se na model primijeni tekstura (slika 38), te model se izvozi u format koji igra poznaje i ubacuje se u level.

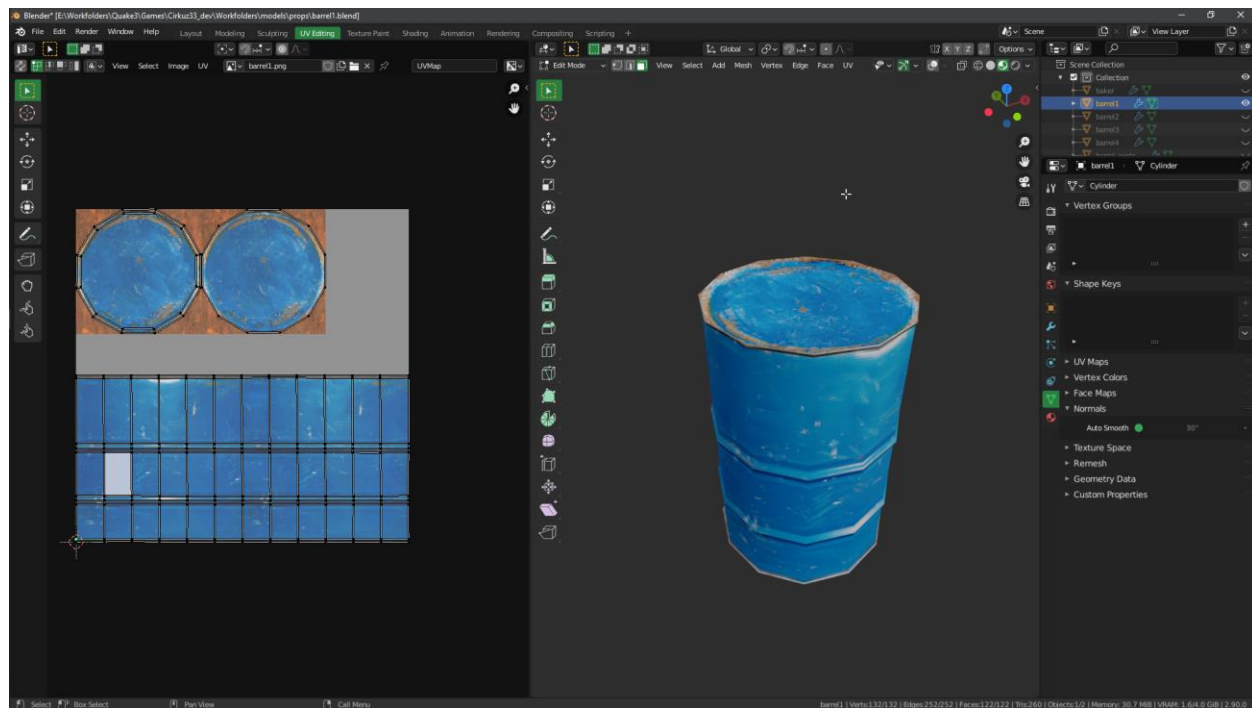
Slike procesa su date na sljedećoj strani.

<sup>7</sup> Primjer 10 retro igara - <https://www.thegamer.com/best-modern-retro-video-games/>

<sup>8</sup> Blender – besplatni i open-source alat za 3D modeliranje - <https://www.blender.org/>



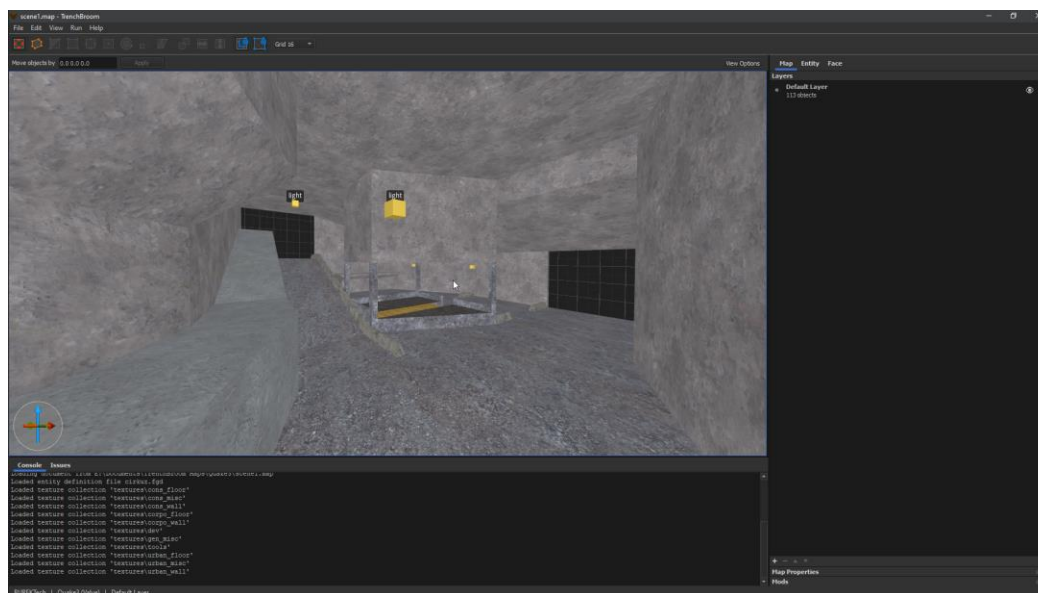
Slika 37. Završeni oblik bačve



Slika 38. Bačva sa primijenjenom teksturom

## 4.3. Izrada levela

Za izradu levela koristi se program *TrenchBroom*<sup>9</sup>. Proces izrade levela je sličan procesu izrade 3D modela, s tim da level editor ima sposobnost dodavanja, uređivanja i brisanja entiteta u level. Primjer levela u *TrenchBroom*-u, kao i rezultat u igri, dat je na slikama ispod:



Slika 39. Prikaz levela u TrenchBroom-u



Slika 40. Prikaz istog u igri

<sup>9</sup> TrenchBroom, besplatan i open-source level editor, Kristian Duske - <https://trenchbroom.github.io/>

## 4.4. Izrada muzike

Za izradu muzike u igri, korišten je program *FL Studio 20*. Koriste se različiti virtuelni instrumenti kao što su električna gitara, bas gitara, harmonika i bubnjevi.

Muzika se spašava u *OGG Vorbis* formatu što dovodi do manje veličine fajlova od *WAV* formata, međutim, za posljedicu ima umjereno povećano vrijeme učitavanja.

## 4.5. Izrada tekstura

Većina tekstura u igri je stvorena pomoću tehnike poznate kao photo-sourcing, odnosno izvlačenje iz slika iz stvarnog svijeta. Proces započinje tako što se uslika bilo koja površina iz stvarnosti (betonski pod, zid, plastična površina itd.).

Zatim se slika manipulira da bi se ispravila perspektiva, te se smanjuje u odgovorne dimenzije, koje moraju biti stepen broja 2 (odgovarajuće vrijednosti su 32, 64, 128, 256, 512, 1024). Pojedine igre sadrže texture dimenzija 2048x2048, 4096x4096, pa čak i 16384x16384 piksela, pomoću tehnike koja je poznata kao *virtual texturing*<sup>10</sup>.

Potom se slika spašava u *PNG* formatu u *textures* folder od igre i spremna je za korištenje.

---

<sup>10</sup> Andreas Neu, „*Virtual Texturing*“, Harvard, Cambridge, maj 2010. - <https://ui.adsabs.harvard.edu/abs/2010arXiv1005.3163N/abstract>

## Zaključak

Dizajn i razvoj igara je vrlo kompleksan i dug proces, međutim vrlo je zabavan i pruža zadovoljstvo onome ko se istim bavi. Razvoj igara se često odvija u ekipama od 5 ili 10 ljudi i više, dok u velikim firmama igara na jednoj igri može raditi preko 150 ljudi.

Premda je korišten game engine iz 1999. godine, uspjeli su se postići znatni rezultati, a sporovi pri radu na igri su bili minimalni, zahvaljujući razvijenim alatima od strane zajednice (*Blender*, *TrenchBroom* itd.) i dokumentaciji istih.

Igra sadrži dva oružja, jednu vrstu neprijatelja i 6 probnih levela. Trenutna veličina fajlova iznosi cca 180 MB, što je čini pogodnom za računare sa malo slobodnog prostora. S obzirom da se koristi *OpenGL 2.1* kao renderer API, igru je moguće pokrenuti na starim i sporim računarima i to bez zapinjanja.

Igra je testirana na laptopu sa procesorom iz *AMD Ryzen 3 5000* serije, sa integrisanom *AMD Radeon Vega 3* grafičkom jedinicom, na kojoj je pri rezoluciji od 1366x768 piksela uspješno radila pri 120 fps.

Projekat je licenciran pod *GPLv2+* licencom i cijeli izvorni kod je dostupan za pregledavanje, uređivanje i korištenje pod uslovima licence.



## Prilog

Izvorni kod igre dostupan je na sljedećem linku:

<https://github.com/Admer456/ioq3-burek/tree/game/cirkuz33>

## Terminologija

**Entitet** – objekat u igri koji vrši određenu funkciju. Postoji više vrsta entiteta kao što su tačkasti entiteti – entiteti koji su ili nevidljivi ili koriste eksterni 3D model – i zapreminski entiteti – entiteti koji za svoj model koriste modele integrisane u level fajlu, koji su stvoreni pomoću level editora.

**fps** – jedna od mjernih jedinica za performansu unutar igre. Označava koliko slika u jednoj sekundi grafička može da proizvede. Ukoliko je broj viši, to znači da igra predstavlja lakši teret za računar, te je manja šansa da dođe do zapinjanja.

**Game engine** – tehnologija koja pokreće igru, izvršava njen kod, učitava njen sadržaj i vrši prikaz istog.

**Igrač** – osoba koja igra igru.

**Level** – igrivi prostor u igri koji može imati cilj i niz zadataka za igrača. Igrač ostvaruje napredak kroz igru tako što rješava zadatke i dolazi do cilja, prelazeći levele.

**Level editor** – program koji se koristi za izradu, najčešće 3D, levela za igre.

**Noda** – nevidljiva tačka u prostoru koja služi za navigaciju NPC-ova.

**NPC** – lik koji nije igrač niti se upravlja od strane igrača (eng. *non-player character*).

**Renderer** – softver ili dio softvera koji koristi grafičku karticu (ili procesor u nekim slučajevima) da bi manifestovao grafike i iste prikazao na ekranu.

**Tekstura** – slika (iz fajla ili generisana) koja se primjenjuje na površine u igri pomoću raznih projekcija za texture.

## Literatura

„**Game Coding Complete 4th Edition**“, Mike McShaffry i David Graham, 2013.  
<https://canvas.projekti.info/ebooks/Game%20Coding%20Complete%20-%204th%20Edition.pdf>

„**Virtual Texturing**“, Andreas Neu, Harvard, Cambridge, maj 2010.  
<https://ui.adsabs.harvard.edu/abs/2010arXiv1005.3163N/abstract>

„**How do bullets work in video games?**“, Tristan Jung, jul 2018.  
<https://medium.com/@3stan/how-do-bullets-work-in-video-games-d153f1e496a8>

„**Quake 3 Source Code Review**“, Fabien Sanglard, jun 2012.  
<https://fabiensanglard.net/quake3/>

(posljednja stranica)

Datum predaje rada: \_\_\_\_\_ 2021. godine

Datum odbrane rada: \_\_\_\_\_ 2021. godine



